arXiv:1109.4460v1 [cs.DS] 21 Sep 2011

A Simple Linear-Space Data Structure for Constant-Time Range Minimum Query*

Stephane Durocher

University of Manitoba, Winnipeg, Canada, durocher@cs.umanitoba.ca

Abstract. We revisit the range minimum query problem and present a new $O(n)$-space data structure that supports queries in $O(1)$ time. Although previous data structures exist whose asymptotic bounds match ours, our goal is to introduce a new solution that is simple, intuitive, and practical without increasing costs for query time or space.

1 Introduction

1.1 Motivation

Along with the mean, median, and mode of a multiset, the minimum (equivalently, the maximum) is a fundamental statistic of data analysis for which efficient computation is necessary. Given a list $A[0 : n-1]$ of n items drawn from a totally orderered set, a *range minimum query (RMQ)* consists of an input pair of indices (i, j) for which the minimum element of $A[i : j]$ must be returned. The objective is to preprocess A to construct a data structure that supports efficient response to one or more subsequent range minimum queries, where the corresponding input parameters (i, j) are provided at query time.

Although the complete set of possible queries can be precomputed and stored using $\Theta(n^2)$ space, practical data structures require less storage while still enabling efficient response time. For all i, if $i = j$, then a range query must report $A[i]$. Consequently, any range query data structure for a list of n items requires $\Omega(n)$ storage space in the worst case [5]. This leads to a natural question: how quickly can an $O(n)$-space data structure answer a range minimum query?

Previous $O(n)$-space data structures exist that provide $O(1)$-time RMQ (e.g., [2–4, 13], see Section 2). These solutions typically require a transformation or invoke a property that enables the volume of stored precomputed data to be reduced while allowing constant-time access and RMQ computation. Each such solution is a conceptual organization of the data into a compact table for efficient reference; essentially, the algorithm reduces to a clever table lookup. In this paper our objective is not to minimize the total number of bits occupied by the data structure (our solution is not succinct) but rather to present a simpler and more intuitive method for organizing the precomputed data to support RMQ efficiently. Our solution combines new ideas with techniques from various

* Work supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

previous data structures: van Emde Boas trees [11], resizable arrays [8], range mode query [17], one-sided RMQ [2], and a linear-space data structure that supports RMQ in $O(\sqrt{n})$ time. The resulting RMQ data structure matches previous optimal bounds of $O(n)$ space and $O(1)$ query time. Our data structure stores efficient representations of the data to permit direct lookup without requiring the indirect techniques employed by previous solutions, such as transformation to a lowest common ancestor query, Cartesian trees, Eulerian tours, and the Four Russians speedup (e.g., [2–4, 13]).

The RMQ problem is sometimes defined such that a query returns only the index of the minimum element instead of the minimum element itself (e.g., [15]). In this paper we require that the actual minimum element be returned. As we discuss in Section 2, several succint data structures exist that support $O(1)$-time RMQ using only $O(n)$ bits of space. In order to return the minimum element in addition to its index, any such data structure must also store the values from the input array A, corresponding to a lower bound of $\Omega(n \log u)$ bits or, equivalently, $\Omega(n)$ words of space in the worst case (analogous lower bounds exist for other array range query problems, e.g., see [5]).

1.2 Definitions, Notation, and Model of Computation

We assume the RAM word model of computation with word size $\Theta(\log u)$, where elements are drawn from a universe $U = \{-u, \dots, u-1\}$ for some fixed $u \geq n$. Unless stated otherwise, memory requirements are expressed in word-sized units. We assume the usual set of $O(1)$-time primitive operations: basic integer arithmetic (addition, subtraction, multiplication, division, and modulo), bitwise logic, and bit shifts. We do not assume $O(1)$-time exponentiation nor, consequently, radicals. When the base operand is a power of two and the result is an integer, however, these operations can be computed using a bitwise left or right shift. All arithmetic computations are on integers in U, and integer division is assumed to return the floor of the quotient. Finally, our data structure only requires finding the binary logarithm of integers in the range $\{0, \dots, n\}$. Consequently, the complete set of values can be precomputed and stored in a table of size $O(n)$ to provide $O(1)$-time reference for the log and $\log\log$ operations at query time, regardless of whether logarithms are included in the RAM model's primitive operations.

A common technique used in array range searching data structures (e.g., [2,17]) is to partition the input array $A[0 : n-1]$ into a sequence of $\lceil n/b \rceil$ blocks, each of size b (except possibly for the last block whose size is $[(n-1) \bmod b]+1$). A query range $A[i : j]$ spans between 0 and $\lceil n/b \rceil$ complete blocks. We refer to the sequence of complete blocks contained within $A[i : j]$ as the *span*, to the elements of $A[i : j]$ that precede the span as the *prefix*, and to the elements of $A[i : j]$ that succeed the span as the *suffix*. See Figure 1. One or more of the prefix, span, and suffix may be empty. When the span is empty, the prefix and suffix can lie either in adjacent blocks, or in the same block; in the latter case the prefix and suffix are equal.

We summarize the asymptotic resource requirements of a given RMQ data structure by the ordered pair $\langle f(n), g(n)\rangle$, where $f(n)$ denotes the storage space it requires and $g(n)$ denotes its worst-case RMQ time. Our discussion focuses primarily on these two measures of efficiency; other measures of interest include the preprocessing time and the update time. Note that similar notation is sometimes used to pair precomputation time and query time (e.g., [2,13]).

2 Related Work

Multiple $\langle \omega(n), O(1)\rangle$ solutions are known, including precomputing RMQs for all query ranges in $\langle O(n^2), O(1)\rangle$, and precomputing RMQs for all ranges of length 2^k for some $k \in \mathbb{Z}^+$ in $\langle O(n \log n), O(1)\rangle$ (Sparse Table Algorithm) [2,13]. In the latter case, a query is decomposed into two (possibly overlapping) precomputed queries. Similarly, $\langle O(n), \omega(1)\rangle$ solutions exist, including the $\langle O(n), O(\sqrt{n})\rangle$ data structure described in Section 3.1.

Several $\langle O(n), O(1)\rangle$ RMQ data structures exist, many of which depend on the equivalence between the range minimum query and lowest common ancestor (LCA) problems. Harel and Tarjan [16] gave the first $\langle O(n), O(1)\rangle$ solution to LCA. Their solution was simplified by Schieber and Vishkin [21]. Berkman and Vishkin [4] showed how to solve the LCA problem in $\langle O(n), O(1)\rangle$ by transformation to RMQ using an Euler tour. This method was simplified by Bender and Farach-Colton [2] to give an ingenious solution which we briefly describe below. Comprehensive overviews of previous solutions are given by Davoodi [9] and Fischer [12], respectively.

The array $A[0 : n-1]$ can be transformed into a Cartesian tree $\mathcal{C}(A)$ on n nodes such that a RMQ on $A[i : j]$ corresponds to the LCA of the respective nodes associated with i and j in $\mathcal{C}(A)$. When each node in $\mathcal{C}(A)$ is labelled by its depth, an Eulerian tour on $\mathcal{C}(A)$ (i.e., the depth-first traversal sequence on $\mathcal{C}(A)$) gives an array $B[0 : 2n-2]$ for which any two adjacent values differ by ± 1. Thus, a LCA query on $\mathcal{C}(A)$ corresponds to a ± 1-RMQ on B. Array B is partitioned into blocks of size $(\log n)/2$. Separate data structures are constructed to answer queries that are contained within a single block of B and those that span multiple blocks, respectively. In the former case, the ± 1 property implies that the number of unique blocks in B is $O(\sqrt{n})$; all $O(\sqrt{n}\log^2 n)$ RMQs on blocks of B are precomputed (the Four Russians technique). In the latter case, a query can be decomposed into a prefix, span, and suffix (see Section 1.2). RMQs on the prefix and suffix are one-sided and can be found in $O(1)$ time (see Section 3.2). The minimum of each block of B is precomputed and stored in $A'[0 : 2n/\log n - 1]$. A RMQ on A' (the minimum value in the span) can be found in $\langle O(n), O(1)\rangle$ using the $\langle O(n' \log n'), O(1)\rangle$ data structure mentioned above due to the shorter length of A' (i.e., $n' = 2n/\log n$).

Fischer and Heun [13] use similar ideas to give a $\langle O(n), O(1)\rangle$ solution to RMQ that applies the Four Russians technique to any array (i.e., it does not require the ± 1 property) on blocks of length $\Theta(\log n)$. Yuan and Atallah [22] examine RMQ on multidimensional arrays and give a new one-dimensional $\langle O(n), O(1)\rangle$

solution that uses a hierarchical binary decomposition of $A[0 : n-1]$ into $\Theta(n)$ canonical intervals, each of length 2^k for some $k \in \mathbb{Z}^+$, and precomputed queries within blocks of length $\Theta(\log n)$ (similar to the Four Russians technique).

When only the index of the minimum is required, Sadakane [20] gives a succinct data structure requiring $4n + o(n)$ bits that supports $O(1)$-time RMQ. Fischer and Heun reduce the space requirements to $2n+o(n)$ [14,15]. Finally, the RMQ problem has been examined in the dynamic setting [7,9], in two and higher dimensions [1, 6, 10, 20, 22], and on trees and directed acyclic graphs [3, 7, 10].

3 A New $\langle O(n), O(1)\rangle$ RMQ Data Structure

The data structure is described in steps, starting with a previous $\langle O(n), O(\sqrt{n})\rangle$ data structure, extending it to $\langle O(n \log\log n), O(\log\log n)\rangle$ by applying the technique recursively, eliminating recursion to obtain $\langle O(n \log\log n), O(1)\rangle$, and finally reducing the space to $\langle O(n), O(1)\rangle$. To simplify the presentation, suppose initially that the input array A has size $n = 2^{2^k}$, for some $k \in \mathbb{Z}^+$; as described in Section 3.5, removing this constraint and generalizing to an arbitrary n is easily achieved without any asymptotic increase in time or space requirements.

3.1 A $\langle O(n), O(\sqrt{n})\rangle$ RMQ Data Structure

The following $\langle O(n), O(\sqrt{n})\rangle$ data structure is known in RMQ folklore (e.g., [19]) and has similar high-level structure to the ± 1RMQ algorithm of Bender and Farach-Colton [2, Section 4]. While subobtimal and often overlooked in favour of more efficient solutions, this data structure forms the basis for our new $\langle O(n), O(1)\rangle$ data structure.

The input array $A[0 : n-1]$ is partitioned into $\sqrt{n}$ blocks of size $\sqrt{n}$. The range minimum of each block is precomputed and stored in a table $B[0 : \sqrt{n}-1]$. See Figure 1. A query range spans between zero and $\sqrt{n}$ complete blocks. The minimum of the span is computed by iteratively examining the corresponding values in B. Similarly, the respective minima of the prefix and suffix are computed by iteratively examining their elements. The range minimum corresponds to the minimum of these three values. Since the prefix, suffix, and array B each contain at most $\sqrt{n}$ elements, the worst-case query time is $\Theta(\sqrt{n})$. The total space required by the data structure is $\Theta(n)$. Precomputation requires only a single pass over the input array in $\Theta(n)$ time. Updates require $\Theta(\sqrt{n})$ time in the worst case; whenever an array element equal to its block's minimum is increased, the block must be scanned to identify the new minimum.

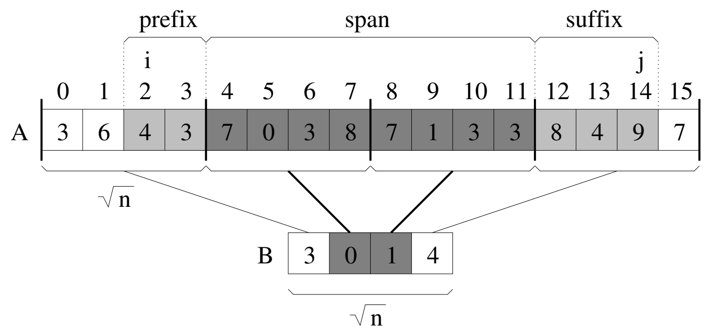

Fig. 1. A $\langle O(n), O(\sqrt{n})\rangle$ data structure: the array A is partitioned into $\sqrt{n}$ blocks of size $\sqrt{n}$. The range minimum of each block is precomputed and stored in array B. A range minimum query $A[2:14]$ is processed by finding the minimum of the respective minima of the prefix $A[2:3]$, the span $A[4:11]$ (determined by examing array B), and the suffix $A[12:14]$. In this example this corresponds to $\min\{3, 0, 4\} = 0$.

3.2 A $\langle O(n \log\log n), O(\log\log n)\rangle$ RMQ Data Structure

One-sided range minimum queries are trivially precomputed [2] and stored in arrays C and C', each of size n, where for each i,

$$C[i] = \begin{cases} \min\{A[i], C[i-1]\} & \text{if } i > 0, \\ A[0] & \text{if } i = 0, \end{cases}$$
$$\text{and} \quad C'[i] = \begin{cases} \min\{A[i], C'[i+1]\} & \text{if } i < n-1, \\ A[n-1] & \text{if } i = n-1. \end{cases} \tag{1}$$

Any subsequent one-sided range minimum query on $A[0:j]$ or $A[j:n-1]$ can be answered in $O(1)$ time by referring to $C[j]$ or $C'[j]$.

The $\langle O(n), O(\sqrt{n})\rangle$ solution discussed in Section 3.1 includes three range minimum queries on subproblems of size $\sqrt{n}$, of which at most one is two-sided. In particular, if the span is non-empty, then the query on array B is two-sided, and the queries on the prefix and suffix are one-sided. Similarly, if the query range is contained in a single block, then there is a single two-sided query and no one-sided queries. Finally, if the query range intersects exactly two blocks, then there are two one-sided queries (one each for the prefix and suffix) and no two-sided queries.

Thus, upon adding arrays C and C' to the data structure, at most one of the three (or fewer) subproblems requires $\omega(1)$ time to identify its range minimum. This search technique can be applied recursively on two-sided queries. By limiting the number of recursive calls to at most one and by reducing the problem size by an exponential factor of $1/2$ at each step of the recursion, the resulting query time is bounded by the following recurrence (similar to that

achieved by van Emde Boas trees [11]):

$$
\begin{aligned}
T(n) &\le \begin{cases} T(\sqrt{n}) + O(1) & \text{if } n > 2, \\ O(1) & \text{if } n \le 2 \end{cases} \\
&\in O(\log\log n).
\end{aligned} \tag{2}
$$

Each step invokes at most one recursive range minimum query on a subarray of size $\sqrt{n}$. Each recursive call is one of two types: i) a recursive call on array B (a two-sided query to compute the range minimum of the span) or ii) a recursive call on the entire query range (contained within a single block).

Recursion can be avoided entirely for determining the minimum of the span (a recursive call of the first type). Since there are $\sqrt{n}$ blocks, $\binom{\sqrt{n}+1}{2} < n$ distinct spans are possible. As is done in the range mode query data structure of Krizanc et al. [17], the minimum of each span can be precomputed and stored in a table D of size n. Any subsequent range minimum query on a span can be answered in $O(1)$ time by reference to table D. Consequently, tables C and D suffice, and table B can be eliminated.

The result is a hierarchical data structure containing $\log\log n{+}1$ levels[1] which we number $0, \ldots, \log\log n$, where the xth level[2] is a sequence of $b_x(n) = n \cdot 2^{-2^x}$ blocks of size $s_x(n) = n/b_x(n) = 2^{2^x}$. See Table 1.

level x	0	1	2	$\ldots$	i	$\ldots$	$\log\log n - 2$	$\log\log n - 1$	$\log\log n$
$b_x(n)$	$n/2$	$n/4$	$n/16$	$\ldots$	$n2^{-2^i}$	$\ldots$	$n^{3/4}$	$\sqrt{n}$	1
$s_x(n)$	2	4	16	$\ldots$	2^{2^i}	$\ldots$	$n^{1/4}$	$\sqrt{n}$	n

Table 1. The xth level is a sequence of $b_x(n)$ blocks of size $s_x(n)$.

Generalizing (1), the new arrays C_x and C'_x are defined by

$$
C_x[i] = \begin{cases} \min\{A[i], C_x[i-1]\} & \text{if } i \ne 0 \bmod s_x(n), \\ A[i] & \text{if } i = 0 \bmod s_x(n), \end{cases}
$$

$$
\text{and} \quad C'_x[i] = \begin{cases} \min\{A[i], C'_x[i+1]\} & \text{if } (i+1) \ne 0 \bmod s_x(n), \\ A[i] & \text{if } (i+1) = 0 \bmod s_x(n). \end{cases}
$$

We refer to a sequence of blocks on level x that are contained in a common block on level $x+1$ as *siblings* and to the common block as their *parent*. Each

[1] Throughout this manuscript, $\log a$ denotes the binary logarithm $\log_2 a$.

[2] Level $\log\log n$ is included for completeness since we refer to the size of the parent of blocks on level x, for each $x \in \{0, \ldots, \log\log n - 1\}$. The only query that refers to level $\log\log n$ directly is the complete array: $i = 0$ and $j = n-1$. The minimum for this singular case can be stored using $O(1)$ space and updated in $O(\sqrt{n})$ time as described in Section 3.1.

block on level $x+1$ is a parent to $s_{x+1}(n)/s_x(n) = s_x(n)$ siblings on level x. Thus, any query range contained in some block at level $x+1$ covers at most $s_x(n)$ siblings at level x, resulting in $\Theta(s_x(n)^2) = \Theta(s_{x+1}(n))$ distinct possible spans within a block at level $x+1$ and $\Theta(s_{x+1}(n) \cdot b_{x+1}(n)) = \Theta(n)$ total distinct possible spans at level $x+1$, for any $x \in \{0, \ldots, \log\log n - 1\}$. These precomputed range minima are stored in table D, such that for every $x \in \{0, \ldots, \log\log n - 1\}$, every $b \in \{0, \ldots, b_{x+1}(n) - 1\}$, and every $\{i, j\} \subseteq \{0, \ldots, s_x(n) - 1\}$, $D_x[b][i][j]$ stores the minimum of the span $A[b \cdot s_{x+1}(n) + i \cdot s_x(n) : b \cdot s_{x+1}(n) + (j + 1)s_x(n) - 1]$.

This gives the following recursive algorithm whose worst-case time is bounded by (2):

Algorithm 1

RMQ(i, j)
if $i = 0$ **and** $j = n - 1$ // query is entire array
    return $minA$ // precomputed array minimum
else
    return $RMQ(\log\log n - 1, i, j)$ // start recursion at top level

RMQ(x, i, j)
if $x > 0$
    $b_i \leftarrow \lfloor i/s_x(n) \rfloor$ // blocks containing i and j
    $b_j \leftarrow \lfloor j/s_x(n) \rfloor$
    if $b_i = b_j$ // i and j in same block at level x
        return RMQ$(x - 1, i, j)$ // two-sided recursive RMQ: $T(\sqrt{n})$ time
    else if $b_j - b_i \geq 2$ // span is non-empty
        $b \leftarrow i \bmod s_{x+1}(n)$
        return $\min\{C'_x[i], C_x[j], D_x[b][b_i + 1][b_j - 1]\}$
        // 2 one-sided RMQs + precomputed span: $O(1)$ time
    else
        return $\min\{C'_x[i], C_x[j]\}$ // 2 one-sided RMQs: $O(1)$ time
else
    return $\min\{A[i], A[j]\}$ // base case (block size ≤ 2): $O(1)$ time

The space required by array D_x for each level $x < \log\log n$ is

$$O\left(s_x(n)^2 \cdot b_{x+1}(n)\right) = O\left(s_{x+1}(n) \cdot b_{x+1}(n)\right) = O(n).$$

Since arrays C_x and C'_x also require $O(n)$ space at each level, the total space required is $O(n)$ per level, resulting in $O(n \log\log n)$ total space for the complete data structure.

For each level $x < \log\log n$, precomputing arrays C_x, C'_x, and D_x is easily achieved in $O(n \cdot s_x(n)) = O(n \cdot 2^{2^x})$ time per level, or $O(n^{3/2})$ total time. Each update requires $O(s_x(n))$ time per level, or $O(\sqrt{n})$ total time per update.

3.3 A $\langle O(n \log\log n), O(1)\rangle$ RMQ Data Structure

Each step of Algorithm 1 described in Section 3.2 invokes at most one recursive call on a subarray whose size decreases exponentially at each step. Specifically, the only case requiring $\omega(1)$ time occurs when the query range is contained within a single block of the current level. In this case, no actual computation or table lookup occurs locally; instead, the result of the recursive call is returned directly (see Line 5 of Algorithm 1). As such, the recursion can be eliminated by jumping directly to the corresponding level of the data structure at which the recursion terminates, that is, the highest level of the data structure for which the query range is not contained in a single block. Any such query can be answered in $O(1)$ time using a combination of at most three references to arrays C and D (see Lines 8 and 10 of Algorithm 1). We refer to the corresponding level of the data structure as the *query level*, whose index we denote by ℓ.

More precisely, Algorithm 1 makes a recursive call whenever $b_i = b_j$, where b_i and b_j denote the respective indices of the blocks containing i and j in the current level (see Line 5 of Algorithm 1). Thus, we seek to identify the highest level for which $b_i \neq b_j$. In fact, it suffices to identify the highest level $\ell \in \{0, \ldots, \log\log n - 1\}$ for which no query of size $j-i+1$ can be contained within a single block. While the query could span the boundary of (at most) two adjacent blocks at higher levels, it must span at least two blocks at all levels less than or equal to ℓ. In other words, the size of the query range is bounded by

$$\begin{aligned}
& & s_\ell(n) <{} & j - i + 1 \leq s_{\ell+1}(n) \\
\Leftrightarrow & & 2^{2^\ell} <{} & j - i + 1 \leq 2^{2^{\ell+1}} \\
\Leftrightarrow & \log\log(j-i+1) - 1 \leq & & \ell < \log\log(j-i+1) \\
\Rightarrow & & & \ell = \lfloor \log\log(j-i) \rfloor.
\end{aligned}$$

As discussed in Section 1.2, since we only require finding binary logarithms of positive integers up to n, these values can be precomputed and stored in a table of size $O(n)$. Consequently, the value ℓ can be computed in $O(1)$ time at query time, where each logarithm is found by a table lookup.

This gives the following simple algorithm whose worst-case running time is constant (note the absence of loops or recursive calls):

Algorithm 2

RMQ(i, j)
if $i = 0$ **and** $j = n - 1$ // query is entire array
  return $minA$ // precomputed array minimum
else if $j - i \geq 2$
  $\ell \leftarrow \lfloor \log\log(j - i) \rfloor$
  $b_i \leftarrow \lfloor i/s_\ell(n) \rfloor$
  $b_j \leftarrow \lfloor j/s_\ell(n) \rfloor$ // blocks containing i and j
  if $b_j - b_i \geq 2$ // span is non-empty
    $b \leftarrow i \bmod s_{\ell+1}(n)$
    return $\min\{C'_\ell[i], C_\ell[j], D_\ell[b][b_i + 1][b_j - 1]\}$
    // 2 one-sided RMQs + precomputed span: $O(1)$ time
  else
    return $\min\{C'_\ell[i], C_\ell[j]\}$ // 2 one-sided RMQs: $O(1)$ time
else
  return $\min\{A[i], A[j]\}$ // query contains ≤ 2 elements

Although the query algorithm differs from Algorithm 1, the data structure remains unchanged except for the addition of precomputed values for logarithms which require $O(n)$ additional space total space. As such, the space remains $O(n \log\log n)$ while the query time is reduced to $O(1)$ in the worst case. Precomputation and update times remain $O(n^{3/2})$ and $O(\sqrt{n})$, respectively.

3.4 A $\langle O(n), O(1) \rangle$ RMQ Data Structure

The data structures described in Sections 3.2 and 3.3 store exact precomputed values in arrays C_x, C'_x, and D_x. That is, for each a and each x, $C_x[a]$ stores $A[b]$ for some b (similarly for C'_x and D_x). If the array A is accessible during a query, then it suffices to store the relative index $b - a$ instead of storing $A[b]$. Thus, $C_x[a]$ stores $b - a$ and the returned value is $A[C_x[a] + a] = A[(b - a) + a] = A[b]$. Since the range minimum is contained in the query range $A[i : j]$ we get that $\{a, b\} \subseteq \{i, \ldots, j\}$ and, therefore,

$$|b - a| \leq j - i + 1 \leq s_{\ell+1}(n).$$

Consequently, for each level x, $\log(s_{x+1}(n)) = 2^{x+1}$ bits suffice to encode any value stored in C_x, C'_x, or D_x. Therefore, for each level x, each table C_x, C'_x, and D_x can be stored using $O(n \cdot 2^{x+1})$ bits. Observe that

$$\sum_{x=0}^{\log\log n - 1} 2^{x+1} < 2 \log n \quad \text{and, similarly,} \quad \sum_{x=0}^{\log\log n - 1} n \cdot 2^{x+1} < 2n \log n. \tag{3}$$

Consequently, the total space occupied by the tables C_x, C'_x, and D_x can be compacted into $O(n \log n)$ bits or, equivalently, $O(n)$ words of space. We now describe how to store this compact representation to enable efficient access.

For each $i \in \{0, \ldots, n-1\}$, the values $C_0[i], \ldots, C_{\log\log n - 1}[i]$ can be stored in two words by (3). Specifically, the first word stores $C_{\log\log n-1}[i]$ and for each $x \in \{0, \ldots, \log\log n - 2\}$, bits $2^{x+1} - 1$ through $2^{x+2} - 2$ store the value $C_x[i]$. Thus, all values $C_0[i], \ldots, C_{\log\log n-2}[i]$ are stored using

$$\sum_{i=0}^{\log\log n-2} 2^{x+1} = \log n - 2 < \log u$$

bits, i.e., a single word, where $\log u$ denotes the word size under the RAM model. The value $C_x[i]$ can be retrieved using a bitwise left shift followed by a right shift or, alternatively, a bitwise logical AND with the corresponding sequence of consecutive 1 bits (all $O(\log\log n)$ bit sequences can be precomputed). An analogous argument applies to the arrays C'_x and D, resulting in $O(n)$ space for the complete data structure.

To summarize, the query algorithm is unchanged from Algorithm 2 and the corresponding query time remains constant, but the data structure's required space is reduced to $O(n)$. Precomputation and update times remain $O(n^{3/2})$ and $O(\sqrt{n})$, respectively. This gives the following lemma:

Lemma 1. *Given any $n = 2^{2^k}$ for some $k \in \mathbb{Z}^+$ and any array $A[0 : n-1]$, Algorithm 2 supports range minimum queries on A in $O(1)$ time using a data structure of size $O(n)$.*

3.5 Generalizing to an Arbitrary Array Size n

To simplify the presentation in Sections 3.1 to 3.4 we assumed that the input array had size $n = 2^{2^k}$ for some $k \in \mathbb{Z}^+$. As we show in this section, generalizing the data structure to an arbitrary positive integer n while maintaining the same bounds on space and time is straightforward.

Let m denote the largest value no larger than n for which Lemma 1 applies. That is,

$$\begin{aligned} && m = 2^{2^{\lfloor \log\log n \rfloor}} \\ \Rightarrow && m \leq n < m^2 \\ \Rightarrow && n/m < \sqrt{n}. \end{aligned} \tag{4}$$

Define a new array $A'[0 : n'-1]$, where $n' = m\lceil n/m \rceil$, that corresponds to the array A padded with dummy data[3] to round up to the next multiple of m. Thus,

$$\forall i \in \{0, \ldots, n'-1\},\ A'[i] = \begin{cases} A[i] & \text{if } i < n \\ +\infty & \text{if } i \geq n. \end{cases}$$

Since $n' = 0 \bmod m$, partition array A' into a sequence of blocks of size m. The number of blocks in A' is $\lceil n/m \rceil < \lceil \sqrt{n} \rceil$.

[3] For implementation, it suffices to store $u - 1$ (the largest value in the universe U) instead of $+\infty$ as the additional values.

By (4) and Lemma 1, for each block we can construct a data structure to support range mode query on that block in $O(1)$ time using $O(m)$ space per block. Therefore, the total space required by all blocks in A' is $O(\lceil n/m \rceil \cdot m) = O(n)$. Construct arrays C, C', and D as before on the top level of array A' using the blocks of size m. The arrays C and C' each require $O(n') = O(n)$ space. The array D requires $O(\lceil n/m \rceil^2) \subseteq O(n)$ space by (4). Therefore, the total space required by the complete data structure remains $O(n)$.

Each query is performed as in Algorithm 2, except that references to C, C', and D at the top level access the corresponding arrays (which are stored separately from C_x, C'_x, and D_x for the lower levels). Therefore, the query time is increased by a constant factor for the first step at the top level, and the total query time remains $O(1)$.

This gives the following theorem:

Theorem 1 (Main Result). *Given any $n \in \mathbb{Z}^+$, and any array $A[0 : n-1]$, Algorithm 2 supports range minimum queries on A in $O(1)$ time using a data structure of size $O(n)$.*

4 Discussion and Directions for Future Work

Succinctness. The data structure presented in this paper uses $O(n)$ words of space. It is not currently known whether its space can be reduced to $O(n)$ bits if a RMQ returns only the index of the minimum element. As suggested by Patrick Nicholson (personal communication, 2011), each array C_x and C'_x can be stored using binary rank and select data structures in $O(n)$ bits of space (e.g., [18]). That is, we can support references to C_x and C'_x in constant time using $O(n)$ bits of space per level or $O(n \log \log n)$ total bits. It is not known whether the remaining components of the data structure can be compressed similarly, or whether the space can be reduced further to $O(n)$ bits.

Higher Dimensions. As shown by Demaine et al. [10], RMQ data structures based on Cartesian trees cannot be generalized to two or higher dimensions. The data structure presented in this paper does not involve Cartesian trees. Although it is possible that some other constraint may preclude generalization to higher dimensions, this remains to be examined.

Dynamic Data. As described, our data structure structure requires $O(\sqrt{n})$ time per update in the worst case. It is not known whether the data structure can be modified to support efficient queries and updates without increasing space.

Acknowledgements

The author thanks Timothy Chan and Patrick Nicholson with whom this paper's results were discussed. The author also thanks the students of his senior undergraduate class in advanced data structures at the University of Manitoba with whom discussion of range searching in arrays inspired him to re-examine the range minimum query problem.

References

1. A. Amir, J. Fischer, and M. Lewenstein. Two-dimensional range minimum queries. In *Proc. CPM*, volume 4580 of *LNCS*, pages 286–294. Springer, 2007.
2. M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. LATIN*, volume 1776 of *LNCS*, pages 88–94. Springer, 2000.
3. M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *J. Alg.*, 57(2):75–94, 2005.
4. O. Berkman and U. Vishkin. Recursive star-tree parallel data structures. *SIAM J. Comp.*, 22(2):221–242, 1993.
5. P. Bose, E. Kranakis, P. Morin, and Y. Tang. Approximate range mode and range median queries. In *Proc. STACS*, volume 3404 of *LNCS*, pages 377–388. Springer, 2005.
6. G. S. Brodal, P. Davoodi, and S. S. Rao. On space efficient two dimensional range minimum data structures. *Algorithmica*, 2011. To appear.
7. G. S. Brodal, P. Davoodi, and S. S. Rao. Path minima queries in dynamic weighted trees. In *Proc. WADS*, volume 6844 of *LNCS*, pages 290–301. Springer, 2011.
8. A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick. Resizable arrays in optimal time and space. In *Proc. WADS*, volume 1663 of *LNCS*, pages 27–48. Springer, 1999.
9. P. Davoodi. *Data Structures: Range Queries and Space Efficiency*. PhD thesis, Aarhus University, 2011.
10. E. Demaine, G. M. Landau, and O. Weimann. On Cartesian trees and range minimum queries. In *Proc. ICALP*, volume 5555 of *LNCS*, pages 341–353. Springer, 2009.
11. P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Proc. Let.*, 6(3):80–82, 1977.
12. J. Fischer. Optimal succinctness for range minimum queries. In *Proc. LATIN*, volume 6034 of *LNCS*, pages 158–169. Springer, 2010.
13. J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem with applications to LCA and LCE. In *Proc. CPM*, volume 4009 of *LNCS*, pages 36–48. Springer, 2006.
14. J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *Proc. ESCAPE*, volume 4614 of *LNCS*, pages 459–470. Springer, 2007.
15. J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comp.*, 40(2):465–492, 2011.
16. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comp.*, 13(2):338–355, 1984.
17. D. Krizanc, P. Morin, and M. Smid. Range mode and range median queries on lists and trees. *Nordic J. Comp.*, 12:1–17, 2005.
18. J. I. Munro. Tables. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *LNCS*, pages 37–42. Springer, 1996.
19. D. Pasailă. Range minimum query and lowest common ancestor. http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor.
20. K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Disc. Alg.*, 5:12–22, 2007.

21. B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comp.*, 17(6):1253–1262, 1988.
22. H. Yuan and M. J. Atallah. Data structures for range minimum queries. In *Proc. SODA*, pages 150–160, 2010.